Lexical Analysis and Parsing

Week 3
CS 212 – Spring 2008

Announcements

- Part 1 (both Compiler & GBA) is due on Friday
- Sections have been split
  - GBA sections
    - There is a GBA section at each section time:
      - M12:20, M7:30, W7:30
    - Location for all GBA sections: Hollister 401
  - Compiler sections
    - Using rooms originally assigned to sections
      - M12:20 Olin Hall 245
      - M7:30 Upson 205
      - M7:30 Upson 205

Compilers

- Basically, a compiler
  - Translates one language (e.g., Java)
  - Into another (e.g., JBC: Java Byte Code)
- Why do this?
  - Idea is to translate a language that is easy for humans to understand into one that is easy for a computer to understand
  - This idea was initially controversial!
- Typical compiler phases
  - Lexical analysis
    - Breaking input into *tokens*
  - Parsing
    - Understanding program's structure
  - Optimization
    - Making the code more efficient (e.g., precomputing constant expressions, avoid recomputing)
  - Code Generation
    - Creating code in a *simpler* language (e.g., JBC, machine code)

Parts of a *Language*

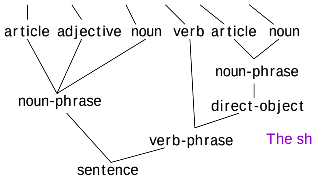
- Human language
  - alphabet → words → sentences → paragraphs → chapters → book
- Computer language
  - alphabet → tokens → statements → program
- Both types of language have
  - Syntax
    - Structural rules
  - Semantics
    - Meaning

Syntax

- Remember diagramming sentences? This was syntax!

  sentence → noun-phrase verb-phrase
  noun-phrase → article [adjective] noun
  verb-phrase → verb direct-object
  direct-object → noun-phrase

- The hungry mouse ate the cheese.



article adjective noun verb article noun
noun-phrase
noun-phrase
direct-object
verb-phrase
sentence

The shiny elbow drank the automobile.

Syntax vs. Semantics

- Syntax = structure
  Semantics = meaning
- Legal syntax does not imply valid meaning
- Examples of semantic rules for a programming language
  - Variables must be declared before use
  - Division by zero causes an error
  - The then-clause is executed only if the if-expression is True
- It's relatively easy to define valid syntax (especially if we get to invent the language)
- It's harder to specify semantics
- How can we specify semantics?
  - Formally, using logic (*axiomatic semantics*)
  - Informally, using explanations in English
  - By reference to a canonical implementation

Compiling Overview

- Compiling a program
  - Lexical analysis
    - Break program into tokens
  - Parsing
    - Analyze token arrangement
    - Discover structure
  - Code generation
    - Create code
- For a computer language, each phase can be completed before the next one begins
- Understanding a sentence
  - Lexical analysis
    - Break sentence into words
  - Parsing
    - Analyze word arrangement
    - Discover structure
  - Understanding
    - Understand the sentence
- For human language, there is feedback between parsing and understanding

Lexical Analysis

- Goal: divide program into *tokens*
- Tokens
  - Individual units or words of a language
  - Smallest element in a language that conveys meaning
  - Examples: operators, names, strings, keywords, numbers
- Tokens can be specified using regular expressions

$a*$ = repeat *a* zero or more times

a^+ = repeat *a* one or more times

[*abc*] = choose one of *a*, *b*, or *c*

. = matches any one character

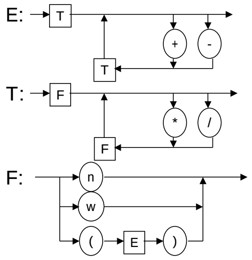
- Examples
  - operator = [+ - * /]
  - integer = $[0123456789]^+$
- For the Compiler Project, we give you the *lexical analyzer* (or *tokenizer*)

Building a Tokenizer

- For tokens, can tell what to do next by checking a few characters (usually 1 character) ahead
  - Example: If it starts with a letter, it's a word; the word ends when you reach a non-alphanumeric character
  - Example: If it starts with a digit, it's a number; if you reach a decimal point, it's a floating point number,...
- Java has a class (introduced in Java 5) java.util.Scanner
  - Can recognize identifiers, numbers, quoted strings, and various comment styles
  - This is more useful than the earlier (Java 1.0) java.io.StreamTokenizer
- Early computer languages were not parsed based on tokens

Specifying Syntax

- How do we specify syntax?
  - Can use a *grammar*
  - Can use a *syntax chart*
- Example grammar
  - (anything in single-quotes is a token; n and w represent a number token and a word token, respectively; parentheses are used for grouping; | indicates choice; * indicates zero-or-more occurrences)
  - E → T (('+' | '-') T)*
  - T → F (('*' | '/ ') F)*
  - F → n | w | '(' E ')'
- Example syntax charts (anything in a rounded box is a token)



Grammars

- The rules in a grammar are called *productions*
- Syntax rules can be specified using a *Context Free Grammar*
  - All productions are of the form V → w
  - V is a single *nonterminal* (i.e., it's not a token)
  - w is word made from *terminals* (i.e., tokens) and nonterminals
- In simple examples, uppercase is used for nonterminals, lowercase for terminals
- Example (ε represents the empty string):

A → ε

A → aAb

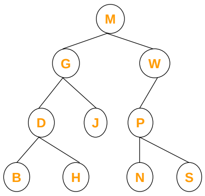
- A grammar defines a *language*
  - Language of example: all strings of the form a^nb^n for $n \geq 0$
- CS 381 for more detail

Building a Parse Tree

- Grammars can be used in two ways
  - A grammar defines a language
  - A grammar can be used to parse a *sentence* (thus, checking if the sentence is *in* the language)
  - For the Compiler Project,
    - We give you the grammar for Bali
    - The *sentence* is a Bali program
- You can show a sentence is in a language by building a *parse tree* (much like diagramming a sentence)
- Example: Show that 8+x/5 is a valid Expression (E) by building a parse tree
  - E → T (('+' | '-') T)*
  - T → F (('*' | '/ ') F)*
  - F → n | w | '(' E ')'

Tree Terminology

- M is the *root* of this *tree*
- G is the *root* of the left *subtree* of M
- B, H, J, N, and S are *leaves*
- P is the *parent* of N
- M and G are *ancestors* of D
- P, N, and S are *descendents* of W
- A collection of trees is called a *??*



Syntactic Ambiguity

- Sometimes a sentence has more than one parse tree

  S → A | aaB
  A → ε | aAb
  B → ε | aB | bB
  - The string aabb can be parsed in two ways
- This kind of ambiguity sometimes shows up in programming languages

if E1 then if E2 then S1 else S2

- This ambiguity actually affects the program's meaning
- How do we resolve this?
  - Provide an extra non-grammar rule (e.g., the *else* goes with the closest *if*)
  - Modify the grammar (e.g., an if-statement must end with a '*fi*')
  - Other methods (e.g., Python uses amount of indentation)
- We try to avoid syntactic ambiguity in Bali

An Extended Example

- A simple computer language
- Each variable is a single letter
- Just two statement types: assignment and do

```
x = 1; y = 1;
do 5:
  x = x * y;
  y = y + 1;
  end;
end.
```

- We can invent a grammar to describe legal programs
  - We need rules for building *expressions*, *statements*, and *programs*
  - We create a *Context Free Grammar* for our simple language

The Grammar

program → statement* **end .**

statement → name **=** expression **;**

statement →
  do expression **:** statement* **end ;**

expression → part [(**+** | **-** | ***** | ***/***) part]

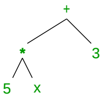
part → (name | *number* | **(** expression **)**)

name → *singleLowercaseLetter*

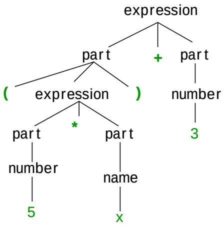
- Notation:
  - * indicates zero or more occurrences
  - [ ] indicates zero or one occurrence
  - (| |) indicates choice
- What is the parse tree for the expression (5 * x) + 3?

Abstract Syntax Tree

- We can build a parse tree, but an AST (*Abstract Syntax Tree*) is more useful
  - Idea is to show less grammar and more meaning



Abstract Syntax Tree

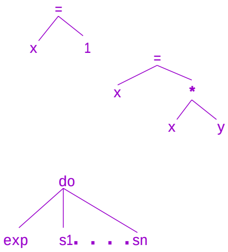


Parse Tree

Designing the AST

- We can invent how the AST should look for each of our language constructs

```
x = 1; y = 1;
do 5:
  x = x * y;
  y = y + 1;
  end;
end.
```



Recursive Descent Parsing

- Idea: Use the grammar to design a recursive program that builds the AST
- To parse a do-statement, for instance
  - We look for each terminal (i.e., token)
  - Each nonterminal (e.g., expression, statement) can handle itself—recursively
- The grammar tells how to write the program

```
public ASTNode parseDo {
   Make sure there is a “do” token;
   exp = parseExpression();
   Make sure there is a “:” token;
   while (not “end” token) {
           s = parseStatement();
           stList.add(s);
           }
   Make sure there is an “end” token;
   Make sure there is a “;” token;
   return DoNode(exp, stList);
}
```

In Practice

- We define a parent class ASTNode
- DoNode can be a subclass
- Each possible node in the AST will have its own subclass of ASTNode
- Some of the grammar’s nonterminals don’t correspond to nodes in the AST
  - E.g., statement, expression, part
- For these we don’t want to create classes
  - But we do need recursive methods to parse these nonterminals

---

Does Recursive Descent Always Work?

- There are some grammars that cannot be used as the basis for recursive descent
  - A trivial example (causes infinite recursion):
    - S -> b
    - S -> Sa
- Can rewrite grammar
  - S -> b
  - S -> bA
  - A -> aA
  - A -> a
- For some constructs Recursive Descent is hard to use
  - Can use a more powerful parsing technique (there are several, but not in this course)

Code Generation

- The same kind of recursive viewpoint can drive our code generation
  - This time we recurse on the AST instead of the grammar
  - Write the code for the root node; the subtrees can take care of themselves

```
class AssignmentStatement extends
    ASTNode {

    String var; ASTNode exp;

    public AssignmentNode (var, exp) {
           this.var = var;
           this.exp = exp;
    }

    public void generate ( ) {
           exp.generate( );
           // Exp result is left on stack
           Generate code to move top
           of stack into mem-location of
           var;
    }
}
```